Table of Contents

Proposal for Handling Coherent Changes in a Configuration

Complete: ▬

Contacts

- **Contacts**: Christopher Jones (FNAL),
- **Hypernews forum**: https://hypernews.cern.ch/HyperNews/CMS/get/edmFramework.html
- **Documentation contact**: Christopher Jones

Purpose of Page

This page presents alternate proposals of handling coherent changes to cmsRun configurations. The presentation is meant to further discussion on the topic.

Introduction

cmsRun configurations are broken into a broad set of processing steps: HLT, GEN, SIM, RawToDIGI, RECO, DQM, Validation, Analysis, etc. However, within a processing step (or even between processing steps) we can have a series of variations. Some variations are differences in processing between data taking periods (*2011 vs 2012 vs 2015*), between machine coditions (*7TeV vs 8TeV* , *50ns vs 25ns* , *40 pileup vs 70 pileup*), between input data (*online vs data vs MC* , *fast sim vs full sim* , *minimum bias data set vs scouting dataset*), standard performance variations (*tight electron cuts vs loose electron cuts*) as well as different possible future detector redesigns. In all these cases we want to share a common setup for the bulk of the configuration and make it easy to apply a perturbation in a standard way which is guaranteed to modify the configuration in a coherent way.

Proposal: Instances of a Modifier class

We could add new class types to the configuration language which are carry out a configuration modification only if requested. These modifiers could operate on a particular object or the entire process. The modifiers would take as input a key/value pair and use that information to decide exactly what should be changed.

Object Modifier

All configuration objects, e.g. `EDProducer` or `Path`, would gain a new `addModifier_` method which would take a function as argument.

```
# Object with default
foo = cms.EDProducer( FooMaker ,bar = cms.int32(6))

# function that handles modification
def _modify_foo(obj, year=0, **kw):
  if 2017 == year:
     obj.bar = 12
foo.addModifier_(_modify_foo)
```

Removal of an object from a configuration could also be supported by having the system pass in a 'remover' method as an argument which the modifier could then use

```
# Object with default
foo = cms.EDProducer( FooMaker )

# function that handles modification
def _modify_foo(obj, year=0, remover=None, **kw):
  if 2017 == year:
     remover()
foo.addModifier_(_modify_foo)
```

NOTE: Does removing an object from a configuration really belong with the object or with a higher level cff?

CFF Modifier

If a more systemic change is needed, like adding new producers to a job, this can be accomplished by creating a `cms.Modifier` object which the Process will load.

```
# standard configuration
...

# function that handles modification
def _modify_stuff(process, year=0, **kw):
  if 2017 == year:
     from Some.Place.fastbar_cfi barFast
     process.bar = barFast

modify_stuff = cms.Modifier(_modify_stuff)
```

Running the modifications

To request a modification to be run, you call the `modifyWith` method of `Process` and pass in the key/value pairs. Process will pass the key/value pairs to each object the Process holds and these objects will execute any modifier they hold. Once that is done the Process will pass the key/value pairs to any cms.Modifier object it has picked up from the configuration.

```
# Do regular setup of process
process.load( Special/Foo/foo_cfi )
...
# switch to a new  version
process.modifyWith(year=2017)
```

Multiple keys

This design supports multiple orthogonal keys

```
# Different subdetectors have different geometries
process.modifyWith(hcalGeom= bold ,
                   trackerGeom= radical ,
                   expectedPileup = 200)
```

The modifiers themselves only have to know about the keys they care about and can ignore the others

```
# Only cares about tracker
def _modify_tracker_geom(obj,process,trackerGeom=None, **kw):
   if trackerGeom is not None:
      ...
```

In the above the default for `trackerGeom` is required to handle the case where that key is not requested by the user. The `**kw` 'eats' any key/value pairs passed to the function for which the function wishes to ignore.

Design attributes

Maintenance

Given that a modifier is required to live in the same file as the item it modifies is declared (or in cff files which are aggregating those items) and we only allow one modifier per item it means the developer is the one who associates key/value to modifications.

If multiple key/value pairs must be composed together to form a 'meta' configuration it could be done by creating a `cms.Modifier` which itself calls `process.modifyWith(...)`.

Discoverability

There is no central place where one can look to determine all possible key/value pairs. However, once an initial configuration is loaded the Process can find all `cms.Modifiers` it holds as well as all modifier functions attached to objects the Process holds. Then using python's introspection it can tell you what keys the different functions are watching. Unfortunately, there is no way the code can find exactly what are the expected values for each key.

Traceability

Once the initial configuration is loaded, the Process object could tell you what objects it holds have modifiers. This allows you to know those objects would be modified if you use a certain key. The Process can also tell you what `cms.Modifier` objects it has collected as well as tell you from which python module the functions the `cms.Modifier` originates.

C++ ParameterSet validation compatibility

It probably would be tricky to modify the C++ ParameterSet validation code to also generate modifiers for the object. Using keys which set values could be accommodated (much as well can generate different _cfi.py files for different module labels), but handling arbitrary operations on values could be hard.

Affect on clones

Given that a modifier is attached to an object instance, it would be easy to apply the same modifier to a clone of the object. There is no need to worry about label changes since the modifier operates directly on the object and does not have to request it from the process.

Possible failures

The `cms.Modifier` objects work directly on the `Process` object and therefore depend on the proper objects having already been attached to the `Process` and on those object having the correct labels.

Proposal: Modifier Singletons

For each configuration variation we could create a new Python class. Developers would import the class into their configuration fragment and attach modification functions to that class. Python treats classes as singletons and guarantees that you can not reassign their names to a different object (unlike regular variables which can always have what they refer to changed).

Declaration

Each configuration variation gets its own Python class. These classes could be declared where ever we'd like. The class would inherit from a class created by the framework group.

```
#This is python module Configuration.StandardSequences.Eras
import FWCore.ParameterSet.Config as cms

class Era2017(cms.ModifierBase):
   pass
```

Modifying an Object

The developer would load the appropriate Modifier class into their cfi file and associate a function to be applied if that Modifier is used

```
from Configuration.StandardSequences.Eras import Era2017

# Object with default
foo = cms.EDProducer( FooMaker ,bar = cms.int32(6))

# function that handles modification
def _modify_foo(obj):
   obj.bar = 12

Era2017.toModify(foo,_modify_foo)
```

We could also support a more explicit syntax

```
from Configuration.StandardSequences.Eras import Era2017

# Object with default
foo = cms.EDProducer( FooMaker ,bar = cms.int32(6))

# state directly what parameters should be changed
Era2017.toModify(foo, bar = 12)
```

Removal of an object could be handled by calling a special function which then remembers you want to remove the object

```
from Configuration.StandardSequences.Eras import Era2017

# Object with default
foo = cms.EDProducer( FooMaker )

Era2017.toRemove(foo)
```

Modifying a Process

If additional modifications are needed, it would be possible to bind an arbitrary function to the Modifier. This function would expect to be passed the `Process` instance.

```
from Configuration.StandardSequences.Eras import Era2017

def _this_does_stuff(process):
   ...

Era2017.toModifyProcess(_this_does_stuff)
```

It is important to keep in mind that the function `_this_does_stuff` will only be bound to `Era2017` if and only if the python file containing that function is loaded into the configuration.

Running the modifications

To request a modification to be run, you import the class into the top level configuration and then pass the `Process` object to the class.

```
# Do regular setup of process
process.load( Special/Foo/foo_cfi )
...
# switch to a new  version
from Configuration.StandardSequences.Eras import Era2017
Era2017.modify(process)
```

Design attributes

Maintenance

Given that the association between the Modifier and the object is done either in the cfi or a cff using that object, it is then the responsibility of the developer to do the association.

If there are specialized Modifiers which must be agregated together to form a 'meta' Modifier, this could be accomplished by passing a Modifier to the 'toModifyProcess' method of another Modifier.

```
#This is python module Configuration.StandardSequences.Eras

import FWCore.ParameterSet.Config as cms
from Configuration.Tracking.Geometries import TrackerLS1_5
...

class Era2017(cms.ModifierBase):
   pass

Era2017.toModifyProcess(TrackerLS1_5)
```

Discoverability

This design allows for but does not require that all Modifier classes be declared in one python file. A user would be responsible to knowing the class name and module for the Modifier they want to apply. That could even be passed as an argument to cmsDriver.py. Given the classes have to inherit from a particular base class, a search for the base class name in LXR should find all options.

Traceability

Once the initial configuration is loaded one can ask the Modifier class which objects have modifiers. One can also use Python's introspection ability to obtain from which python module the functions were obtained. If we also include specialized class functions to the Modifier class (such as the `toRemove` function) we could even know the intent of the change. If the explicit syntax were used the system could print out exactly the values which would change. Even without the explicit syntax we could pass in a dummy object which could gather the key/values for the functions operating on the objects but not those operating on the process (unless we could also fake a dummy process instance).

C++ ParameterSet validation compatibility

Having the C++ ParameterSet validation system generate the needed python code in the autogenerated cfi files seems straight forward. The developer would have to give the full python module and python class name for the Modifiers and then we'd use code similar to how we deal with multiple labels.

Affect on clones

If 'toModify' method of the Modifier also told the object to which Modifier it was attached and what function to apply, then any clone function could be updated to also tell the Modifier it was changed. By also remember the function to apply it means even if a developer used python's own 'deepcopy' module to do the cloning (which we have seen people use) the code would still work. When the Modifier is handed a process it would loop through all the held objects and if the object thought it was attached to the Modifier, the Modifier would execute the associated function.

Possible failures

Any process based function attached to a Modifier via `toModifyProcess` works directly on the `Process` object and therefore depend on the proper objects having already been attached to the `Process` and on those objects having the correct labels.

Proposal: Jean-Roch Vlimant's

There is one python class for which all modifier function are attached. There is a separate python dictionary which maps a key to a list of modifier functions held by the python class instance. Finally we'd need a function which is passed a Process and a key and then executes the list of funcitions.

Declaration

The ModifierHolder class is declared in some standard python package. The dictionary which relates functions to keys could be declared in the same or different package.

```
#This is python module Configuration.StandardSequences.Modifier
import FWCore.ParameterSet.Config as cms

class Modifier(object):
   pass

#the _ keeps python from importing this into other modules
_namesToModifiers = { "2015" : "tracking_2015+ecal_original+...",
                     "2017" : "...",
                     "HighPileup" : "...",
                     ...
                   }

def runModifiers(process,key):
   for funcs in _namesToModifiers.get(key,list()):
      for f in funcs.split("+"):
         if hasattr(Modifier,f):
            getattr(Modifier,f)(process)
```

variation 1

Instead of holding the function names in one large string, it would probably be easier to hold them in a python tuple

```
_namesToModifiers = { "2015" : ("tracking_2015","ecal_original", ...),
                     "2017" : ("...", ...),
                     "HighPileup" : ("...", ...) ,
                     ...
                   }

def runModifiers(process,key):
   for funcs in _namesToModifiers.get(key,list()):
      for f in funcs:
         if hasattr(Modifier,f):
            getattr(Modifier,f)(process)
```

In the above, there is no guarantee that the strings used to declare the name of a function bound to the `Modifier` class actually correspond to any methods of that class, either becuase of a typo in the string or because the python file which attached the function to the `Modifier` was not imported anywhere in the configuration.

variation 2

Given one has to know at the top level all possible function names that exist, one could instead just import all modules which declare functions into one top level module and have a python dictionary holding keys to list of functions.

```
from RecoTracker.MadeUpModule.some_cfi import modify_foo
from RecoMuon.AnotherName.other_cfi import modify_bar
...
_namesToModifiers = { "2015" : (modify_foo, modify_bar, ...),
                     "2017" : (...),
                     "HighPileup" : (...) ,
                     ...
                   }
def runModifiers(process,key):
   for funcs in _namesToModifiers.get(key,list()):
      for f in funcs:
         f(process)
```

Given you are using the actual functions there is no way to have a typo. The downside is since every possible modifier function for a given key was loaded and these modifiers modify the `Process` and not individual objects then this variant is much more susceptible to failures due to loading only part of a configuration into a `Process`. That is they are more susceptible to an object not having been added to a `Process`.

Modifying an Object

The developer would load the Modifier class into their cfi file and associate a function to be applied if that Modifier is used

```
from Configuration.StandardSequences.Modifier import Modifier

# Object with default
foo = cms.EDProducer( FooMaker ,bar = cms.int32(6))

# function that handles modification
def _modify_foo(process):
   #this function could be called on a process that doesn't have 'foo'
   if hasattr(process,"foo"):
      process.foo.bar = 12

Modifier.modify_foo = _modify_foo
```

Given that the object names are globally declared, it would be possible to have the modify function not apply to the process but instead directly to the global object

```
from Configuration.StandardSequences.Modifier import Modifier

# Object with default
foo = cms.EDProducer( FooMaker ,bar = cms.int32(6))

# function that handles modification
#    process argument is ignored
def _modify_foo(process):
   global foo
   foo.bar = 12

Modifier.modify_foo = _modify_foo
```

Modifying a Process

Since the modifier functions are handed the full Process, they can do any arbitrary modification

```
from Configuration.StandardSequences.Modifier import Modifier

def _this_does_stuff(process):
   ...
```

```
Modifier.this_does_stuff = _this_does_stuff
```

Running the modifications

To request a modification to be run, you import the runModifiers function into the top level configuration and then pass the `Process` object and key to the function.

```
# Do regular setup of process
process.load( Special/Foo/foo_cfi )
...
# switch to a new  version
from Configuration.StandardSequences.Modifier import runModifiers
runModifiers(process,"Era2017")
```

Design attributes

Maintenance

All additional functions require the maintainer of the `runModifiers` function to update the key to function name dictionary. This is presumable a Level 2 job.

Discoverability

The list of known keys can be found by reading the key to function name dictionary.

Traceability

To first order, it is impossible to know what objects will be modified by an given key. The best one can do is find the function names associated to the key and then for each function name look to see if they were bound to the `Modifier` class in this process. If they are bound then one could use python's reflection to determine which python module declared the function. If a modifier function has no conditional statements, one could probe what objects it attempts to modify by passing in a class which is a dummy of a Process.

C++ ParameterSet validation compatibility

I don't have any good idea of how this could be supported.

Affect on clones

Given that the functions operate on a `Process` and not a configuration object a new function would have to be written for each clone.

Possible failures

Given that the modifier functions works directly on the `Process` object and therefore depend on the proper objects having already been attached to the `Process` and on those objects having the correct labels.

Contrasting the Proposals

The Modifier instances provides key/value pairs so in principal the value could be used to pass user specified quantities to a series of modules. This might allow an alternative method of dealing with PAT customization. The Modifier singleton design would not support such a use.

The keys used in the Modifier instance proposal are arbitrary and it is up to the developer to exactly match the name of the key to which he/she is trying to associate. It is therefore possible to have an uncaught typo. The Modifier singleton is bound to a python class name and a typo would cause a python runtime error.